CDF/MEMO/STATISTICS/PUBLIC/6850

Detecting a Bad Random Number Generator

Joel Heinrich—University of Pennsylvania

January 22, 2004

1 Introduction

The main purpose of this note is to exhibit a defect in the Linux C and C++ standard library pseudo-random number generator `rand` and the Linux/UNIX system-library generator `random`. Since Linux `rand` and Linux/UNIX `random` use identical algorithms, we will use "`random`" to refer to both. The defect is uncovered when `random` fails a simple empirical test.

However, in the process of writing this note, a **major bug** in CLHEP's `RandEngine` class (for Linux platforms) was discovered, which is described in **section 4**. This bug is unrelated to the failure of `random` mentioned above, since we call `random` directly, rather than using `RandEngine`, **which is simply broken** on Linux platforms.

Empirical tests of random number generators involve "goodness-of-fit" issues, and this note can also serve as a simple example of goodness-of-fit. (It is not intended as an exhaustive validation of available generators, since only a single test is employed.)

Good uniform random number generators must produce deviates that are:

- Uniformly distributed.
- Independent.

"Independent" means that knowing the values of previously produced deviates provides no extra information about the values of the subsequent deviates. As non uniformity (in one dimension) is easily detected, "bad" uniform random number generators, in practice, actually fail a test of independence.[1]

One can view "independent" as meaning "uniformly distributed in an N-dimensional hypercube" for arbitrarily large N. For N not too large, for example, one could test for independence by:

1. divide the hypercube into 10^N bins,
2. generate $M \gg 10^N$ random points $(u_1, u_2, \ldots u_N)$ in N-space
3. use the generated points to populate the bins

[1] However, see section 4 for a generator that actually does fail the uniformity test.

4. calculate $\chi^2 = \sum_{i=1}^{10^N} \frac{(B_i - M10^{-N})^2}{M10^{-N}}$, where B_i is the population of the ith bin

5. compute the P-value: the probability of obtaining $\geq$ the observed χ^2 (here for $10^N - 1$ DOF)

6. reject the generator if the P-value is too small

Although the choice of 10 above is just for illustration, it is clear that this scheme is not practical for more than of order 10^7 bins. For large N, we need another idea.

Unfortunately, for large N, there is no simple scheme that is guaranteed to catch all bad generators—goodness of fit in $N \gg 1$ dimensions is an unsolved problem. The best we can do is calculate a "statistic", which is a projection from N dimensions to a smaller number of dimensions—usually just one dimension. The statistic we employ in this note is the "maximum spacing".

2 The Maximum Spacing Statistic

Generating $n-1$ uniform deviates in the interval 0 to 1 divides that interval into n segments. The largest of these we will call the "maximum spacing", and denote it's value by $\mathcal{S}$. Clearly, we have $1/n \leq \mathcal{S} \leq 1$. The smallest case, $\mathcal{S} = 1/n$, occurs when the $n-1$ deviates, sorted in ascending order, have the values $1/n, 2/n, \ldots, (n-1)/n$, dividing the unit interval into n segments of equal size. The largest, $\mathcal{S} = 1$, occurs when the uniform deviates are just 0's and 1's, so that the unit interval remains undivided.

The calculation of $\mathcal{S}$ is very well adapted to the C++ STL algorithms. Given that generator `rangen()` returns an `int` in the range 0 to `INT_MAX`, the following C++ fragment suffices to calculate the maximum spacing:

```
w[0] = 0;  // array w must be able to hold n+1 int's
std::generate(w+1,w+n,rangen);
w[n] = INT_MAX;
std::sort(w+1,w+n);
std::adjacent_difference(w,w+n+1,w);
double S = *std::max_element(w+1,w+n+1)/(INT_MAX+1.0);
```

An important point is that the statistic $\mathcal{S}$ is independent of the *order* of the $n-1$ generated deviates: the first step after generating is to sort them. That is, this statistic looks at *what* values occur in a set of a certain size, and not the order in which they are added to the set. A related test, George Marsaglia's "birthday spacings" test [1, 2], also has this property.

What makes the maximum spacing statistic interesting is that the cumulative distribution function for $\mathcal{S}$ is known exactly[3]:

$$P(x) = \text{prob}(\mathcal{S} > x) = \sum_{k=1}^{\lfloor \frac{1}{x} \rfloor} (-1)^{k-1} \binom{n}{k} (1-kx)^{n-1} \qquad (x > 0)$$

where $\lfloor t \rfloor$ stands for "the greatest integer less than or equal to t". (Note that $\binom{n}{k} = 0$ when $k > n$.) Since the identity[4]

$$0 = \sum_{k=0}^{n} (-1)^{k-1} \binom{n}{k} (1-kx)^{n-1}$$

holds for all x, we also have

$$P(x) = 1 - \sum_{k=\lceil \frac{1}{x} \rceil}^{n} (-1)^{k-1} \binom{n}{k} (1-kx)^{n-1} \qquad (x > 0)$$

where $\lceil t \rceil$ stands for "the least integer greater than or equal to t".

Thus, it is easy to compute $P(x)$: Depending on the value of x, one chooses the formula which requires the fewest terms, so no more than $n/2$ terms are needed. Although the sum involves terms of opposite signs, the cancellation is not severe (providing the above rule is followed), so computing the sum in double precision is adequate.

Another reason that the maximum spacing is interesting as a test for uniform random number generators is that similar tests of the past have focused more the minimum. For example, the "minimum distance test" and the "3D-spheres test" from George Marsaglia's Diehard battery of tests[2] both look at minimum distances (in 2 and 3 dimensions, respectively).

3 Performing the Test

The procedure is the following:

1. Generate N uniform random deviates, compute $\mathcal{S}$ and $U = P(\mathcal{S})$. Note that U itself should behave as a uniform random deviate.

2. Repeat step-1 10^6 times, collecting the U deviates in a histogram with 10 bins.

3. Compute $\chi^2 = 10^{-5} \sum_{i=1}^{10} (B_i - 10^5)^2$, where B_i represents the contents of the ith bin.

4. Compute the P-value for the observed χ^2, which has 9 DOF.

This is done for $N = 10, 20, 30, \ldots, 200$, for 11 different random number generators. The resulting P-value are shown in the following table:

N	1	2	3	4	5	6	7	8	9	10	11
10	0.322	0.693	0.729	0.996	0.85	0.000658	0.8	0.485	0.732	0.936	0.176
20	0.0332	0.823	0.0181	0.364	0.899	0.000248	0.091	0.698	0.861	0.518	0.996
30	0.159	0.742	0.177	0.0834	0.333	0.0872	0.831	0.0335	0.102	0.245	0.551
40	0.00173	0.443	0.575	0.0753	0.358	0.188	0.899	0.533	0.495	0.629	0.661
50	2.35e-18	0.869	0.00112	0.687	0.734	0.613	0.706	0.792	0.66	0.33	0.711
60	2.09e-12	0.284	0.823	0.375	0.708	0.715	0.739	0.105	0.627	0.897	0.388
70	3.4e-14	0.503	0.455	0.16	0.548	0.409	0.901	0.593	0.659	0.939	0.491
80	7.21e-16	0.923	0.211	0.0099	0.633	0.795	0.787	0.153	0.217	0.436	0.0507
90	7.14e-12	0.0891	0.057	0.33	0.819	0.832	0.932	0.127	0.245	0.347	0.251
100	9.23e-12	0.841	0.714	0.428	0.221	0.347	0.838	0.0128	0.936	0.97	0.95
110	2.49e-08	0.888	0.295	0.879	0.61	0.468	0.636	0.343	0.0465	0.568	0.384
120	2.01e-11	0.13	0.356	0.654	0.68	0.604	0.678	0.735	0.759	0.894	0.999
130	1.72e-06	0.559	0.622	0.164	0.259	0.59	0.505	0.716	0.608	0.96	0.0445
140	4.16e-13	0.118	0.444	0.256	0.0882	0.827	0.672	0.666	0.45	0.446	0.472
150	5.73e-06	0.489	0.125	0.376	0.0525	0.125	0.29	0.388	0.407	0.134	0.722
160	7.51e-05	0.976	0.491	0.0771	0.679	0.856	0.491	0.859	0.75	0.43	0.4
170	3.8e-08	0.729	0.238	0.305	0.175	0.154	0.113	0.429	0.992	0.834	0.0586
180	2.3e-05	0.179	0.121	0.994	0.825	0.0427	0.479	0.979	0.0264	0.486	0.053
190	0.053	0.33	0.693	0.338	0.504	0.9	0.766	0.923	0.343	0.0358	0.656
200	0.0174	0.281	0.504	0.424	0.082	0.327	0.143	0.698	0.858	0.992	0.827

The numbers at the head of each column identify a particular uniform random number generator. The P-values should also behave as uniform deviates. Obviously, this is not true for generator 1, and generator 6 is also suspect. The 11 generators are described in the following subsections, whose numbers correspond to the column headers.

3.1 `random`

This generator is provided by the system on Linux, Sun/UNIX and SGI/UNIX platforms: it comes from the standard Berkeley source. It was considered to be a superior generator, the best of the system supplied ones. It is therefore surprising that it performs badly on this test, for $N >\sim 10$. When a generator fails a test in this way, it is automatic grounds for rejection: don't use it for any purpose. (The test above was run on Linux, but I have confirmed that `random` also fails as badly on Sun and SGI.)

The C and C++ standard library generator `rand` is quite inadequate on the Sun and SGI platforms, as they only provide 16 bits of precision. For Linux, there was an attempt to correct this situation by using the `random` algorithm, which was believed to be of high quality, for `rand`.

Since `rand` is inadequate on all 3 platforms, it also must be rejected: Don't use the C or C++ standard library `rand` for any purpose on any platform. Numerical Recipes[5] gives the same advice, and colorfully comments:

> If all scientific papers whose results are in doubt because of bad `rand()`'s were to disappear from library shelves, there would be a gap on each shelf about as big as your fist.

3.2 `random` with the Lüscher fix

Martin Lüscher's scheme to fix generators like `random` is to generate a small block of random deviates, then "throw away" a large block, causing the generator to "forget" about the small block. Column 1 suggests that $N = 10$ is safe, so this plan is implemented as follows:

```
int randomlux() {
  static int n=0;
  if(++n>=10) {
    for(int j=0;j<90;++j) random();
    n=0;
  }
  return random();
}
```

As column 2 shows, this scheme does actually correct the defect present in `random`—at least as far as the maximum spacing test is concerned.

3.3 `lrand48`

`lrand48` is also provided by UNIX/LINUX. It generates uniform random deviates using a linear congruential algorithm with 48-bit integer arithmetic. It passes this test, as shown in column 3. Strangely, the Linux man page says:

> These [`lrand48`] functions are declared obsolete by SVID 3, which states that `rand`(3) should be used instead.

which is NOT good advice. `lrand48` has the interesting feature of allowing the user to change its default multiplier via a call to `lcong48`, which gives the user a chance to really mess things up.

3.4 multiply with carry

Column 4 is George Marsaglia's "multiply with carry" generator, which passes. Quoting Marsaglia: "I predict it will be the system generator for many future computers". Unfortunately, this prediction has not yet come true.

3.5 VAX system generator

A sentimental favorite, this linear congruential generator (column 5) also passes the maximum spacing test.

3.6 `randu`

The very bad generator of column 6 was supplied by most systems of the 1960's to unsuspecting users. Interestingly, it does fail the maximum spacing test, but in contrast to `random`, it is bad in the neighborhood of $N \simeq 20$, and recovers for $N >\sim 40$. `randu` fails many other tests, and was abandoned long ago.

3.7 `rndm`

This obsolete generator (column 7) from CERNLIB passes the test. It apparently has a rather short period by modern standards.

3.8 `ranmar`

Column 8 also passes. According to the CERNLIB documentation: "The period is about 10^{43} and the quality is good but it fails some tests." I interpret this to mean "Don't use it".

3.9 `ranlux`

Column 9, `ranlux` operating with the default settings, passes. Probably the best of the CERNLIB generators, it uses the Lüscher scheme to fix a generator that is known to fail some tests. By calling the configuration routine `rluxgo`, the user can switch off the fix, thereby converting it into a bad generator.

3.10 `ranecu`

Column 10's `ranecu` is yet another CERNLIB generator, and passes the maximum spacing test. Probably not as good as `ranlux` at it's default setting.

3.11 `ranarray`

The latest version of Don Knuth's `ranarray`[6] generator in column 11 passes the test. It also uses the Lüscher scheme to rehabilitate a generator that is known to fail some tests.

4 A `random` Horror Story

The CLHEP class library[7] provides a class `RandEngine` that is advertised as

Simple random engine using `rand()` and `srand()` functions from C standard library to implement the `flat()` basic distribution and for setting seeds.

`HepDouble RandEngine::flat()`
It returns a pseudo random number between 0 and 1, according to the standard stdlib random function `rand()` but excluding the end points.

Let's look at CLHEP's implementation of `RandEngine::flat()`:

```
double RandEngine::flat()
{
   // rand() is such a horrible generator, it only generates 15 bits.
   // The quick fix here to get it to at LEAST 32 bits is to grab two values,
   // and concatenate them, but that still leaves two bits empty and these
   // we grab from the number of times flat() has been called, i.e. seq.

   return double( (unsigned int)(        (rand() << 17) |   // bits 31-17
                                   (seq++ & 0x3  << 15) |   // bits 16,15
                                                 rand()     // bits 14-0
                                 ) * mantissa_bit_32
                  );
}
```

The implementer has realized that 15 bits are not enough, and calls `rand()` twice to get more bits. Unfortunately, the Linux `rand()` actually provides 31 bits, and the `RandEngine::flat()` code shown above makes no provision for that. The ISO standard only guarantees that `RAND_MAX` will be *at least* $2^{15} - 1$, so the implementer has made an unjustified assumption based on `rand()`'s behavior on a specific platform.

Deviates produced by `RandEngine::flat()` on Linux are not even uniform. Referring to the code above, while bit 31 is random, bits 30–15 are produced by an "|" operation between random bits, which means that, for each of bits 30–15, there is a 75% probability that it is 1.

We can calculate the exact mean of the Linux `RandEngine::flat()` deviates as the sum of the contributions from each bit:

$$\text{mean} = \frac{1}{4} + \frac{3}{16} + \frac{3}{32} + \frac{3}{64} + \frac{3}{128} + \cdots = \frac{5}{8} - 2^{-19} \simeq 0.625$$

where -2^{-19} is the correction for the fact that it's only bits 30–15 that are "enriched", not all the bits 30–0.

To test this prediction, we run the following trivial C++ program on a Linux platform

```
#include<iostream>
#include "CLHEP/Random/RandEngine.h"
int main() {
  RandEngine r;
  const int nsum=10000000;
  double sum=0;
  for(int i=0;i<nsum;++i)
    sum += r.flat();
  std::cout << sum/nsum << std::endl;
  return 0;
}
```

On Linux, the result printed is 0.624956, as the reader can easily verify (the statistical error here is of order 0.0001, so the -2^{-19} term is negligible). This, of course, **is a disaster**, as uniform deviates obviously should have a mean of $\frac{1}{2}$.

While this particular bug might be fixed by using "`^`" instead of "`|`", the fact that the C and C++ `rand()` algorithm can be different on every platform, and can change at every system upgrade, argues that using `rand` is too dangerous.

As of the date of this note, this bug still exists in all Linux versions of libCLHEP.a used at CDF. The extent of problems caused by it is unknown. The bug was found (just a few hours ago) by reading the source file `RandEngine.cc`, not by running an empirical test.

5 Conclusions

Most modern generators pass the maximum spacing test, and the infamous `randu` fails it. The UNIX/Linux `random` generator fails badly, and should not be used. The C and C++ standard library generator `rand` is the same as `random` on Linux platforms, and `rand` should not be used on any platform. CLHEP's `RandEngine::flat()` provides an example of what can go wrong if this rule is violated. Providing defective generators is an old tradition among system programmers, and is likely to continue until the `rand` algorithm is specified by international standards.

I have tried to keep a sense of humor while writing this note, but unfortunately, I will also have to change generators used in some of the software I have written, if I am to follow my own recommendations.

References

[1] Donald E. Knuth, "The Art of Computer Programming", Volume 2, 3rd edition, (Addison-Wesley, Reading, Massachusetts, 1998), §3.3.2, page 71.

[2] `stat.fsu.edu/~geo/diehard.html`
`stat.fsu.edu/pub/diehard/`

[3] H. A. David and H. N. Nagaraja, "Order Statistics", 3rd edition, (Wiley, Hoboken New Jersey, 2003), §6.4, page 135.

[4] Donald E. Knuth, "The Art of Computer Programming", Volume 1, 3rd edition, (Addison-Wesley, Reading, Massachusetts, 1997), §1.2.6, page 64, equation 34.

[5] William H. Press, et al., "Numerical Recipes", 2nd edition, (Cambridge University Press, Cambridge, 1992), §7.1
`lib-www.lanl.gov/numerical/bookcpdf/c7-1.pdf`

[6] `www-cs-faculty.stanford.edu/~knuth/news02.html#rng`
`www-cs-faculty.stanford.edu/~knuth/programs.html#rng`

[7] CLHEP—A Class Library for High Energy Physics
`wwwasd.web.cern.ch/wwwasd/lhc++/clhep/`
`wwwasd.web.cern.ch/wwwasd/lhc++/clhep/manual/RefGuide/Random/RandEngine.html`